CS 4110

Programming Languages & Logics

Lecture 35
Typed Assembly Language

1 December 2014

Announcements

- Foster Office Hours today 4-5pm
- Optional Homework 10 out
- Final practice problems out this week

Schedule

One and a half weeks ago...

- Typed Assembly Language

Last Monday

- Polymorphism
- Stack Types

Today

- Compilation

Certified Compilation

An *automatic* way to generate certifiable low-level code

Type safety implies memory, security properties

Certified Compilation

An *automatic* way to generate certifiable low-level code

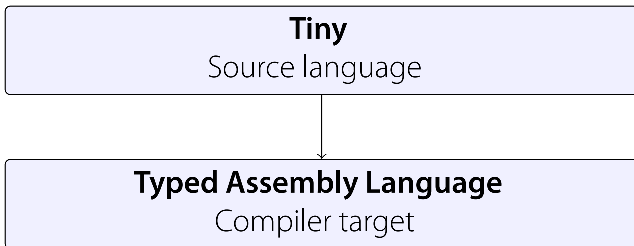
Type safety implies memory, security properties

Tiny
Source language

Certified Compilation

An *automatic* way to generate certifiable low-level code

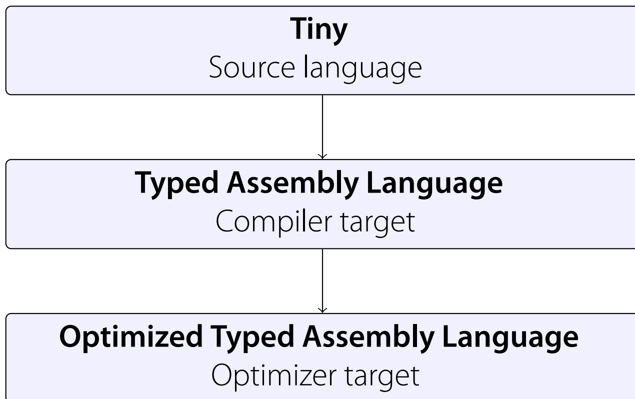
Type safety implies memory, security properties



Certified Compilation

An *automatic* way to generate certifiable low-level code

Type safety implies memory, security properties



The Tiny Language

Features

- Integer expressions
- Conditionals
- Recursive functions
- Function pointers (but no closures)
- A strong, static type system

The Tiny Language

Features

- Integer expressions
- Conditionals
- Recursive functions
- Function pointers (but no closures)
- A strong, static type system

Example program

letrec fun *fact* (*n*:int) : int =
    if $n = 0$ then 1 else n * *fact* ($n - 1$)
in *fact* (42)

Tiny Type System

Expressions

$$\Phi \vdash x : \Phi(x)$$

Tiny Type System

Expressions

$$\Phi \vdash x : \Phi(x) \qquad \Phi \vdash f : \Phi(f)$$

Tiny Type System

Expressions

$$\Phi \vdash x : \Phi(x) \qquad \Phi \vdash f : \Phi(f) \qquad \Phi \vdash n : \mathrm{int}$$

Tiny Type System

Expressions

$$\Phi \vdash x : \Phi(x) \qquad \Phi \vdash f : \Phi(f) \qquad \Phi \vdash n : \text{int}$$

$$\frac{\Phi \vdash e_1 : \text{int} \qquad \Phi \vdash e_2 : \text{int}}{\Phi \vdash e_1 + e_2 : \text{int}}$$

Tiny Type System

Expressions

$$\Phi \vdash x : \Phi(x) \qquad \Phi \vdash f : \Phi(f) \qquad \Phi \vdash n : \text{int}$$

$$\frac{\Phi \vdash e_1 : \text{int} \qquad \Phi \vdash e_2 : \text{int}}{\Phi \vdash e_1 + e_2 : \text{int}} \qquad \frac{\Phi \vdash e_1 : \tau_1 \to \tau_2 \qquad \Phi \vdash e_2 : \tau_1}{\Phi \vdash e_1\ e_2 : \tau_2}$$

Tiny Type System

Expressions

$$\Phi \vdash x : \Phi(x) \qquad \Phi \vdash f : \Phi(f) \qquad \Phi \vdash n : \text{int}$$

$$\frac{\Phi \vdash e_1 : \text{int} \qquad \Phi \vdash e_2 : \text{int}}{\Phi \vdash e_1 + e_2 : \text{int}} \qquad \frac{\Phi \vdash e_1 : \tau_1 \to \tau_2 \qquad \Phi \vdash e_2 : \tau_1}{\Phi \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Phi \vdash e_1 : \text{int} \qquad \Phi \vdash e_2 : \tau \qquad \Phi \vdash e_3 : \tau}{\Phi \vdash \text{if } e_1 = 0 \text{ then } e_2 \text{ else } e_3 : \tau}$$

Tiny Type System

Expressions

$$\Phi \vdash x : \Phi(x) \qquad \Phi \vdash f : \Phi(f) \qquad \Phi \vdash n : \text{int}$$

$$\frac{\Phi \vdash e_1 : \text{int} \qquad \Phi \vdash e_2 : \text{int}}{\Phi \vdash e_1 + e_2 : \text{int}} \qquad \frac{\Phi \vdash e_1 : \tau_1 \to \tau_2 \qquad \Phi \vdash e_2 : \tau_1}{\Phi \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Phi \vdash e_1 : \text{int} \qquad \Phi \vdash e_2 : \tau \qquad \Phi \vdash e_3 : \tau}{\Phi \vdash \text{if } e_1 = 0 \text{ then } e_2 \text{ else } e_3 : \tau}$$

$$\frac{\Phi \vdash e_1 : \tau_1 \qquad \Phi, x : \tau_1 \vdash e_2 : \tau_2}{\Phi \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_1}$$

Tiny Type System

Declarations

$$\frac{\Phi, x : \tau_1 \vdash e : \tau_2}{\Phi \vdash \text{fun } f(x : \tau_1) : \tau_2 = e : (f : \tau_1 \to \tau_2)}$$

Tiny Type System

Declarations

$$\frac{\Phi, x : \tau_1 \vdash e : \tau_2}{\Phi \vdash \text{fun } f(x : \tau_1) : \tau_2 = e : (f : \tau_1 \to \tau_2)}$$

Programs

$$\frac{\begin{array}{c}\Phi = f_1 : \tau_{11} \to \tau_{12}, \ldots, f_k : \tau_{k1} \to \tau_{k2} \\ \Phi \vdash d_i : (f_i : \tau_{i1} \to \tau_{i2}) \qquad \Phi \vdash e : \text{int}\end{array}}{\vdash \text{letrec } d_1 \cdots d_k \text{ in } e}$$

Tiny Type System

Declarations

$$\frac{\Phi, x : \tau_1 \vdash e : \tau_2}{\Phi \vdash \text{fun } f(x : \tau_1) : \tau_2 = e : (f : \tau_1 \to \tau_2)}$$

Programs

$$\frac{\begin{array}{c}\Phi = f_1 : \tau_{11} \to \tau_{12}, \ldots, f_k : \tau_{k1} \to \tau_{k2} \\ \Phi \vdash d_i : (f_i : \tau_{i1} \to \tau_{i2}) \qquad \Phi \vdash e : \text{int}\end{array}}{\vdash \text{letrec } d_1 \cdots d_k \text{ in } e}$$

Exercise: Check that *fact* is well typed

Type-preserving Compilation

- Most compilers consist of a series of transformations
- In our compiler, each step will propagate types
- A transformation consists of two sub-translations:
  - From source types to target types
  - From source terms to target terms
- After each step, can typecheck the intermediate output code to detect errors in the compiler
- Key correctness property will be that transformations preserve types (as well as semantics)

Type Translation

$\mathcal{T}[\![\cdot]\!]$ maps Tiny types to TAL Types

Type Translation

$\mathcal{T}[\![\cdot]\!]$ maps Tiny types to TAL Types

- Integers: $\mathcal{T}[\![\mathsf{int}]\!] = \mathsf{int}$

Type Translation

$\mathcal{T}[\![\cdot]\!]$ maps Tiny types to TAL Types

- Integers: $\mathcal{T}[\![\text{int}]\!] = \text{int}$
- Functions:
  - Caller pushes argument and return address on stack
  - Callee pops the return address and argument and places result in r_a

$$\mathcal{T}[\![\tau_1 \to \tau_2]\!] = \forall \rho.\{sp : \mathcal{K}[\![\tau_2, \rho]\!] :: \mathcal{T}[\![\tau_1]\!] :: \rho\} \to \{\}$$

where

$$\mathcal{K}[\![\tau, \sigma]\!] = \{sp : \sigma, r_a : \mathcal{T}[\![\tau]\!]\} \to \{\}$$

$$\mathcal{K}[\![\tau]\!] = \{r_a : \mathcal{T}[\![\tau]\!]\} \to \{\}$$

Expression Translation

$\mathcal{E}[\![\cdot]\!]$ maps Tiny expressions (really typing derivations) to labeled code blocks

Expression Translation

$\mathcal{E}[\![\cdot]\!]$ maps Tiny expressions (really typing derivations) to labeled code blocks

- We use the stack extensively:
  - To evaluate an expression, evaluate subexpressions then push them onto the stack
  - Return final value in r_a
  - M maps expression variables to stack offsets
  - $I(M)$ increments the offset associated with each variable in M
- Overall, uses just two registers, r_a and r_t, and the stack
- Shape of the translation is $\mathcal{E}[\![e]\!]_{M,\sigma} = J$, where J is a sequence of typed, labeled blocks.
- Write L_f for the TAL label of function f

Expression Translation

Expression variables

$$\mathcal{E}[\![x]\!]_{M,\sigma} = \text{sld}\ r_a, M(x)$$

Expression Translation

Expression variables

$$\mathcal{E}[\![x]\!]_{M,\sigma} = \mathsf{sld}\ r_a, M(x)$$

Function variables

$$\mathcal{E}[\![f]\!]_{M,\sigma} = \mathsf{mov}\ r_a, L_f$$

Expression Translation

Expression variables

$$\mathcal{E}[\![x]\!]_{M,\sigma} = \text{sld } r_a, M(x)$$

Function variables

$$\mathcal{E}[\![f]\!]_{M,\sigma} = \text{mov } r_a, L_f$$

Integers

$$\mathcal{E}[\![n]\!]_{M,\sigma} = \text{mov } r_a, n$$

Expression Translation

Expression variables

$$\mathcal{E}[\![x]\!]_{M,\sigma} = \text{sld } r_a, M(x)$$

Function variables

$$\mathcal{E}[\![f]\!]_{M,\sigma} = \text{mov } r_a, L_f$$

Integers

$$\mathcal{E}[\![n]\!]_{M,\sigma} = \text{mov } r_a, n$$

Addition

$$\begin{aligned}\mathcal{E}[\![e_1 + e_2]\!]_{M,\sigma} = {} & \mathcal{E}[\![e_1]\!]_{M,\sigma} \\ & \text{push } r_a \\ & \mathcal{E}[\![e_2]\!]_{I(M),\text{int} :: \sigma} \\ & \text{pop } r_t \\ & \text{add } r_a, r_t, r_a\end{aligned}$$

Expression Translation

Application

$$
\begin{aligned}
\mathcal{E}[\![e_1\ e_2]\!]_{M,\sigma} = {} & \mathcal{E}[\![e_1]\!]_{M,\sigma} \\
& \text{push } r_a \\
& \mathcal{E}[\![e_2]\!]_{l(M),\mathcal{T}[\![\tau_1 \to \tau_2]\!] :: \sigma} \\
& \text{pop } r_t \\
& \text{push } r_a \\
& \text{push } L_r[\rho] \\
& \text{jmp } r_t[\sigma] \\
L_r : {} & \forall \rho.\ \mathcal{K}[\![\tau_2, \sigma]\!]
\end{aligned}
$$

where $\Phi \vdash e_1 : \tau_1 \to \tau_2$
and L_r fresh

Expression Translation

Conditional

$$
\begin{aligned}
\mathcal{E}[\![\text{if } e_1 = 0 \text{ then } e_2 \text{ else } e_3]\!]_{M,\sigma} &= \mathcal{E}[\![e_1]\!]_{M,\sigma} \\
&\quad \text{bneq } r_a, L_{else}[\rho] \\
&\quad \mathcal{E}[\![e_2]\!]_{M,\sigma} \\
&\quad \text{jmp } L_{end}[\rho] \\
L_{else} &: \forall\rho.\{sp : \sigma\} \to \{\} \\
&\quad \mathcal{E}[\![e_3]\!]_{M,\sigma} \\
&\quad \text{jmp } L_{end}[\rho] \\
L_{end} &: \forall\rho.\mathcal{K}[\![\tau, \sigma]\!]
\end{aligned}
$$

where $\Phi \vdash e_2 : \tau$ and $\Phi \vdash e_2 : \tau$
and L_{else} and L_{end} fresh

Declaration Translation

Function

$$
\begin{aligned}
\mathcal{F}[\![\text{fun } f\,(x : \tau_1) : \tau_2 = e]\!] = L_f :\ & \mathcal{T}[\![\tau_1 \rightarrow \tau_2]\!] \\
& \mathcal{E}[\![e_2]\!]_{x\,:=\,2, \mathcal{K}[\![\tau_2, \rho]\!]\,::\,\mathcal{T}[\![\tau_1]\!]\,::\,\rho} \\
& \text{pop } r_t \\
& \text{sfree } 1 \\
& \text{jmp } r_t
\end{aligned}
$$

Program Translation

Program

$$
\begin{aligned}
\mathcal{P}[\![\text{letrec } d_1 \cdot d_k \text{ in } e]\!] = {} & \mathcal{F}[\![d_1]\!] \\
& \vdots \\
& \mathcal{F}[\![d_k]\!] \\
L_{main} : {} & \forall\rho.\{sp : \mathcal{K}[\![\text{int}, \rho]\!] :: \rho\} \to \{\} \\
& \mathcal{E}[\![e]\!]_{\cdot, \mathcal{K}[\![\text{int}, \rho]\!] :: \rho} \\
& \text{pop } r_t \\
& \text{jmp } r_t
\end{aligned}
$$

To run the program, push return address on stack and jump to L_{main}

Result is returned in r_a

Factorial

L_{fact}: $\forall\rho.\ \{sp : \mathcal{K}[\![\text{int}]\!] :: \text{int} :: \rho\}$
    sld r_a,2
    bneq r_a,$L_{else}[\rho]$
    mov r_a,1
    jmp L_{end}
L_{else}: $\forall\rho.\ \{sp : \mathcal{K}[\![\text{int}]\!] :: \text{int} :: \rho\}$
    sld r_a,2
    push r_a
    mov r_a,L_{fact}
    push r_a
    sld r_a,4
    push r_a
    mov r_a,1
    pop r_t
    sub r_a,r_t,r_a
    pop r_t
    push $L_r[\rho]$
    jmp $r_t[\text{int} :: \mathcal{K}[\![\text{int}, \rho]\!] :: \text{int} :: \rho]$

Factorial

L_{fact}: $\forall\rho.\ \{sp : \mathcal{K}[\![\text{int}]\!] :: \text{int} :: \rho\}$
sld r_a,2
bneq r_a,$L_{else}[\rho]$
mov r_a,1
jmp L_{end}
L_{else}: $\forall\rho.\ \{sp : \mathcal{K}[\![\text{int}]\!] :: \text{int} :: \rho\}$
sld r_a,2
push r_a
mov r_a,L_{fact}
push r_a
sld r_a,4
push r_a
mov r_a,1
pop r_t
sub r_a,r_t,r_a
pop r_t
push $L_r[\rho]$
jmp $r_t[\text{int} :: \mathcal{K}[\![\text{int}, \rho]\!] :: \text{int} :: \rho]$

L_r: $\forall\rho.\ \{sp : \mathcal{K}[\![\text{int}]\!] :: \text{int} :: \rho, r_a : \text{int}\}$
pop r_t
mul r_a,r_t,r_t
jmp $L_{end}[\rho]$
L_{end}: $\forall\rho.\ \{sp : \mathcal{K}[\![\text{int}]\!] :: \text{int} :: \rho, r_a : \text{int}\}$
pop r_t
sfree 1
jmp r_t

Compiler properties

Theorem

If $\vdash P$ *then there exists* Ψ *such that* $\vdash \mathcal{P}[\![P]\!] : \Psi$

Proof by induction on P...

Main idea: We don't have to trust the compiler because we can typecheck the assembly code emitted as output!

Optimizations

Almost any compiler will produce better code than ours! (But how many compilers fit on four slides?)

The type system makes it possible to implement many standard optimizations to generate better code:

- Instruction selection and scheduling
- Register allocation
- Common subexpression elimination
- Redundant load/store elimination
- Strength reduction
- Loop-invariant removal
- Tail-calls
- and others...

Types do not interfere with the most common optimizations

Tail-call Optimization

- A crucial optimization for functional languages
- Applies when the final operation in a function *f* is a call to another function *g*
- Instead of having *f* push the return address onto the stack and engage in the normal calling sequence, *f* pops all of its temporary values, jumps directly to *g* and never returns!

Tail-call Optimization

Unoptimized code

L_f: $\forall\rho.\ \{sp : \mathcal{K}[\![\tau, \rho]\!] :: \tau_f :: \rho, r_a : \tau_g\} \to \{\}$

```
    ...
    salloc 2            % setup stack frame
    sst L_r             % push return address
    sst r_a,2           % push argument
    jmp L_g[τ :: τ_f :: ρ]
```

L_r: $\forall\rho.\ \{sp : \tau :: \tau_f :: \rho, r_a : \tau\} \to \{\}$

```
    pop r_t             % pop return address
    sfree 1             % discard f's argument
    jmp r_t             % return
```

Optimized code

L_f: $\forall\rho.\ \{sp : \mathcal{K}[\![\tau, \rho]\!] :: \tau_f :: \rho, r_a : \tau_g\} \to \{\}$

```
    ...
    sst r_a,2
    jmp L_g[ρ]    % g returns to f's caller
```